

COMPILADORES - 21780 Práctica final



Universitat
de les Illes Balears

Curso 2024-2025

Sergio Garat Estelrich
Alejandro Rodríguez
Álvaro Pimentel
Alexander Ortega

Índice

Introducción	4
Características del lenguaje	4
Tecnologías utilizadas	5
El lenguaje de programación	6
Análisis léxico	8
Proceso y clases generadas	8
Token.java	8
lexico.flex y Lexico.java	9
Output: Tokens.txt y lexicalError.txt	10
Análisis sintáctico	11
Proceso y clases generadas	12
sintactico.cup, Parser.java y ParserSym.java	12
Output: syntaxError.txt	13
Análisis semántico	14
Tabla de símbolos	14
Variables de instancia	14
Funciones principales	15
Gestión de errores	16
Clase CompilerError	16
Funcionalidades clave:	16
Clase SymTabError	17
Generación de código intermedio	18
Código de tres direcciones (C3@)	18
Conjunto de instrucciones	18
Clase BackTables	19
Clase Var	20
Clase Proc	20
Clase Etiq	20
Generación de código ensamblador	21
Lenguaje empleado	21
Estructura del programa	21
Cabecera (header)	21
Bloque principal	21
Bloque final (footer)	22
Optimizaciones código ensamblador	22
Conclusión	23
Anexos	24
Anexo 1: Gramática del lenguaje	24
Anexo 2: Ejemplos de programas	28
Ejemplos exitosos	28
Ejemplos erróneos	29
Anexo 3: Guía de uso	29
Anexo 4: Vídeo explicativo	31

Introducción

El trabajo realizado ha consistido en la creación de un compilador desde cero para un lenguaje de programación propio, abarcando todas sus fases, desde el análisis léxico hasta la generación de código ensamblador.

En esta memoria se describen las distintas fases del proceso, así como la solución implementada y los desafíos que han ido surgiendo.

Características del lenguaje

Nuestro compilador es capaz de reconocer y operar con las siguientes características:

❖ Elementos esenciales

- Cuerpo principal 'main' y definición de métodos/funciones (solamente soporta que las funciones se definan antes de la función 'main')
- Definición y uso de funciones con parámetros
- Declaración y uso de tipos de datos:
 - Enteros
 - Booleanos
- Operaciones:
 - Asignación
 - Condicional: if, elif, else
 - Bucles: 'while' y 'for'
 - Llamada y retorno de funciones
- Funcionalidades de entrada por teclado y salida por pantalla
- Operadores soportados:
 - Aritméticos (2): "+", "-"
 - Relacionales (2): "<", ">"
 - Lógicos (2): "&&", "||"

❖ Elementos adicionales

- Nuevos tipos de datos: Cadenas de texto (String)
- Operadores adicionales:
 - Aritméticos: "*", "/", "%"
 - Relacionales: "<=", ">=", "==", "!="
 - Lógico: "!"
- Las funciones soportan ser recursivas (ver ejemplo 1)

❖ **Tabla de símbolos avanzada y sistema predictivo de gestión de errores**

- La tabla de símbolos se explica con más detalle en secciones posteriores
- El sistema de gestión de errores cubre 4 categorías principales:
 - Léxicos
 - Sintácticos
 - Semánticos
 - Relativos a la tabla de símbolos

Tecnologías utilizadas

El equipo, basándose en sus conocimientos previos, ha optado por utilizar Java para el desarrollo del compilador. Nuestro equipo domina este lenguaje a la perfección y, además, resulta ser lo suficientemente potente y flexible para cumplir con nuestros objetivos. Asimismo, este lenguaje ofrece las bibliotecas necesarias para la construcción de nuestro compilador. En particular, hemos decidido emplear dos herramientas específicas:

- ❖ JFlex, herramienta útil para crear el código en java del Scanner, encargado del análisis léxico.
- ❖ CUP, herramienta utilizada juntamente con JFlex para generar el análisis sintáctico del compilador

Además, como código ensamblador hemos decidido utilizar GAS (GNU Assembler) que es el ensamblador del proyecto GNU y es una herramienta que convierte el código fuente en código máquina (binario) que puede ser ejecutado por la CPU.

El lenguaje de programación

El lenguaje de programación presentado es de tipo imperativo y fuertemente tipado, permitiendo la declaración explícita de variables mediante ':' seguido de su tipo, con asignaciones usando ':='. Soporta estructuras de control condicional como 'if-elif-else' y bucles 'for' y 'while'. Permite la declaración de funciones con un valor de retorno, y junto a ello, permite la recursividad. Cuenta con una función principal como punto de entrada y utiliza read/print para la entrada/salida de datos. Su sintaxis es clara y estructurada, facilitando la programación modular y el manejo de operaciones aritméticas y condicionales de manera intuitiva.

En la siguiente imagen se muestra la asignación de valores a variables en el lenguaje de programación. Se declaran tres variables, 'num', 'str' y 'b', una de cada tipo permitido en el lenguaje, utilizando los dos puntos seguido del tipo para indicar el tipo de dato. La asignación de valores se realiza mediante el operador ':=', lo que indica que num recibe el valor 100, str "HOLA MUNDO!" y b es 'true'. Este tipo de asignación permite inicializar variables con valores específicos en el momento de su declaración.

```
num : number := 100;
str : string := "HOLA MUNDO!";
b : boolean := true;
```

Ahora vemos como se presenta la definición de una función. La del ejemplo es una función recursiva para calcular el factorial de un número pasado por parámetro. Dentro de la función, se declara la variable result con un valor inicial de 0, y posteriormente se le asigna el resultado de la llamada recursiva. Finalmente, la función retorna el resultado del factorial de 'num'. La sintaxis utilizada para definir funciones incluye la palabra clave function, seguida del nombre de la función, los parámetros entre paréntesis con sus respectivos tipos, y el tipo de retorno después de los dos puntos.

```
function factorial(num : number) : number {
    result : number := 0;
    if (num == 0) {
        result := 1;
    } else {
        result := num * factorial(num - 1);
    }
    return result;
}
```

A continuación, vemos una llamada a la función factorial con el argumento fact. El resultado de la función es pintado por pantalla mediante la función print(). Esto significa que la función

es evaluada con el valor actual de fact, y su resultado se pinta por pantalla. La llamada a funciones se realiza mediante el nombre de la función, seguido de paréntesis, dentro de los cuales se pasan los argumentos requeridos, respetando el orden y tipo de datos definidos en la función.

```
print(factorial(fact));
```

Finalmente, observamos un pequeño fragmento con distintas sentencias:

```
//probando funcion recursiva factorial//
print("Inserta un número (entre 0 y 15) para calcular el factorial: ");
fact: number := read();
print("Número insertado: ");
print(fact);
print("\n");

if ((fact >= 0) && (fact <= 15)){
    print(fact);
    print("!= ");
    print(factorial(fact));
    print("\n");
} else {
    print(fact);
    print(" es demasiado grande.\n");
}
```

Con este último ejemplo, cabe comentar que se hace uso de muchos print distintos porque el lenguaje no permite las operaciones de las cadenas de caracteres. Es decir, no permite lo que podemos observar en otros lenguajes como java, concatenación de valores string y numeros (System.out.println("Hola mundo!" + num);)

Análisis léxico

Para llevar a cabo esta primera fase del proceso de compilación, es necesario definir los elementos fundamentales que compondrán nuestro programa y que el compilador podrá identificar. Estos elementos vendrán definidos en la clase Token.java. Cualquier elemento que se encuentre en un futuro programa de entrada y no esté contemplado en esta clase (como enumeración) será considerado un error.

En nuestro programa, además de reconocer los tokens definidos, toda la información recopilada durante esta etapa se almacena en un archivo de texto llamado Tokens.txt, en el que también se registran los posibles errores detectados.

Junto con la clase Token.java, es necesario trabajar con otro archivo denominado *lexico.flex*, donde se definen todos los elementos estructurados que serán introducidos por el usuario, así como las reglas que rigen una determinada entrada. Un ejemplo sencillo de esto es la gestión de comentarios: si el compilador detecta la secuencia '//', ignorará todo el contenido que se encuentre dentro de ella.

Una vez establecida la prioridad de los elementos definidos en el archivo *lexico.flex*, mediante el método JFlex.Main.generate(), si el proceso se realiza correctamente, se generará automáticamente la clase Lexico.java gracias a la herramienta JFLEX. Esta clase contendrá toda la información necesaria para llevar a cabo el análisis léxico de un archivo de texto de manera automatizada.

Proceso y clases generadas

Durante el análisis léxico se han utilizado y posteriormente generado varios archivos:

Token.java

Este primer archivo contiene la clase Token, la cual se utiliza como estructura de datos para el análisis de cada uno de los conjuntos de caracteres introducidos por el usuario. Un token es un objeto con los atributos id, line, column y lexeme. El atributo **id**, es un valor de entre los especificados en el enumerado Tokens, el cual contiene todos y cada uno de los tokens que podemos encontrar (palabras reservadas, operadores, identificadores, etc.). Los atributos '**line**' y '**column**' guardan la posición del texto en la que se encuentra el token

(posición del primer carácter del token). Por último, el atributo '**lexeme**' sirve para guardar el contenido del token.

Dentro de esta clase encontramos también el método toString() que nos ayuda a pintar los tokens, con un formato adecuado, en el archivo de salida.

lexico.flex y Lexico.java

Para generar las reglas y patrones del análisis léxico, se hace uso de la herramienta JFlex, la cual permite escribir un archivo .flex (con una estructura simple) y lo convierte en el archivo .java ejecutable.

Dentro de lexico.flex primero debemos definir funciones y código en JAVA que será necesario para la gestión del análisis léxico. Dentro de este apartado indicamos el nombre de la clase a generar (Lexico.java), los atributos y la función 'writeToken' que nos ayudará a escribir los tokens determinados en el archivo de salida Tokens.txt.

A continuación definimos los patrones de palabras o conjunto de caracteres, que posteriormente serán identificados como tokens. Palabras reservadas, como while o for; separadores, como la coma, o el punto y coma; la forma de los identificadores, primero obligatoriamente una letra seguida de cualquier conjunto de caracteres; o los operadores, entre otros.

Por último, debemos decir, que debe hacer el analizador al encontrar cada uno de los tokens. Para cada uno de los tokens que encontremos se realizará lo siguiente:

1. Primero crearemos el objeto Token con los atributos correspondientes
2. Nos ayudaremos de la función antes definida para escribir el token dentro del archivo de salida
3. Por último, retornamos un símbolo (ver más adelante, en el análisis sintáctico) para que posteriores pasos hagan uso de este.

Mostramos el ejemplo del token IF:

```
{INST_IF} {
    Token token = new Token(Tokens.INST_IF,yyline,yycolumn, yytext());
    writeToken(token);
    return symbol(Tokens.INST_IF.name(), ParserSym.inst_if);
}
```


En cada uno de los patrones que se han estipulado se realiza lo mismo, devolviendo un símbolo distinto dependiendo del Token reconocido.

Output: Tokens.txt y lexicalError.txt

Tras completar el análisis léxico correctamente se genera el archivo Tokens.txt, el cual contiene todos los tokens analizados. En cambio, si hay algún error léxico en el análisis, este se escribirá dentro del archivo lexicalError.txt

Análisis sintáctico

Para realizar una correcta validación de un programa, un analizador léxico por sí solo no es suficiente. La siguiente etapa del proceso de compilación es el análisis sintáctico. En esta fase, se verifica que la secuencia de instrucciones proporcionada por el usuario sea, al menos, estructuralmente correcta.

Esto se puede comprender fácilmente con el siguiente ejemplo: en el análisis sintáctico, por motivos de claridad y legibilidad, se ha optado por dividir el proceso en dos partes principales: `lexico.flex` y `sintactico.cup`.

En la primera, de forma similar al análisis léxico, se definen las reglas que el analizador podrá interpretar. La segunda, en cambio, es donde se encuentran los aspectos esenciales de nuestro lenguaje, es decir, su gramática. ¿Cómo determina nuestro compilador el orden en el que deben descomponerse las instrucciones? ¿Cómo reconocer cuándo ha encontrado un elemento considerado “final”?

La respuesta es sencilla: esto se logra a través de los símbolos definidos en el archivo `.cup`, los cuales se dividen en dos categorías principales. Los símbolos terminales, que permiten identificar estructuras completas, y los símbolos no terminales, que permiten establecer las reglas de composición del lenguaje.

A continuación (ver Anexo 1), se presenta la gramática del lenguaje desarrollado, donde los elementos escritos en mayúsculas corresponden a los símbolos terminales, mientras que los escritos en minúsculas representan los símbolos no terminales.

Cada una de las producciones tienen acciones definidas a realizar cuando estas se producen. En todas y cada una de ellas se realiza una serie de acciones, acabando con el retorno de un símbolo, el correspondiente en cada caso. Las demás acciones, como comprobaciones de tipo, o gestión de etiquetas y demás, pertenecen al análisis semántico.

Para la gramática, principalmente se ha hecho uso de un enfoque descendente recursivo (LL). Lo podemos ver en ejemplos como las producciones de ‘instructions’, que hacen uso de recursividad por la izquierda. Es cierto, que para decir que se ha hecho uso de un analizador LL(1), se deberían eliminar estas recursiones por la izquierda para evitar bucles infinitos. Además, la estructura modular de la gramática (inicialización, declaración de

funciones, instrucciones, etc.), es una característica más de este enfoque de parseo descendente.

Proceso y clases generadas

Para el análisis sintáctico se ha hecho uso de la herramienta y librerías de CUP, el cual mediante el archivo `sintactico.cup`, genera una diversidad de clases java, para proceder con el análisis.

`sintactico.cup`, `Parser.java` y `ParserSym.java`

Así como en el análisis léxico utilizamos la herramienta JFlex, junto a un archivo `.flex` para generar la clase correspondiente, en esta etapa del compilador se hace uso de CUP, que junto a un archivo `.cup`, genera las clases correspondientes para continuar con el análisis sintáctico.

El archivo `.cup` tiene una estructura similar a un archivo `.flex`, primero se define en código JAVA, las funciones y atributos que vamos a necesitar en esta clase, posteriormente se declaran los símbolos, terminales y no terminales, que se van a utilizar; y por último, se elabora la gramática, con el conjunto de reglas necesario para hacer que nuestro compilador reconozca correctamente las estructuras aceptadas.

En este, primero inicializamos los atributos y funciones necesarios, en este caso, necesitamos objetos que a posteriori se utilizarán para la gestión semántica, la generación de código intermedio y ensamblador y la gestión de errores. Las funciones inicializadas son para la gestión de errores sintácticos y la inicialización de tipos.

A continuación, se definen todos los símbolos que se van a utilizar en la posterior gramática. Cada uno de estos símbolos pertenecen a una clase JAVA determinada, generada a mano en la carpeta "symbols".

Por último, se elabora la gramática (Anexo 1) la cual contiene todas las producciones necesarias. Cada una de estas, tiene sus reglas, y las acciones a realizar. Durante el vídeo se explican con un poco más de profundidad estas reglas.

Tras acabar con la definición del archivo `.cup`, si no hay errores, la herramienta genera los archivos `Parser.java` y `ParserSym.java`. En la clase `ParserSym` se han generado todos los

símbolos terminales. La clase Parser, será la encargada de realizar todas las acciones que hemos definido en cada una de las reglas de producción de la gramática.

Output: `syntaxError.txt`

Durante el análisis sintáctico, a pesar de generar variables, escribir en la tabla de símbolos, generar código intermedio, etc. el único output que deriva explícitamente del análisis sintáctico es el fichero de errores `syntaxError.txt`. Este fichero solamente aparecerá, si se detecta un error sintáctico en el programa introducido por el usuario.

Análisis semántico

Este es el último paso del proceso de front-end, y es el encargado de que toda la información escrita en forma de código sea correcta. Los identificadores se utilizan según su significado.

Durante el análisis semántico ocurrirá lo siguiente:

- Generación de código intermedio
- Gestión de la memoria utilizada a lo largo de la ejecución.

Todo este proceso se realiza mediante la tabla de símbolos.

Tabla de símbolos

La tabla de símbolos se encarga de gestionar y almacenar la información de las variables, funciones y parámetros declarados en el código fuente, organizándose según su ámbito de visibilidad. La tabla de símbolos organiza el código según los ámbitos y asegura que las declaraciones sean válidas y consistentes, permitiendo el seguimiento de funciones y variables durante la ejecución del compilador. Su estructura es la siguiente:

Variables de instancia

- `int scope`: Indica el ámbito actual.
- `ArrayList<Integer> scopeTable`: Guarda índices que indican dónde podemos escribir en `expansionTable` para cada ámbito.
- `HashMap<String, Descriptor> descriptionTable`: Almacena identificadores visibles en el ámbito actual junto con su información (nombre, tipo, ámbito, primer parámetro si es función).
- `ArrayList<ExpandInfo> expansionTable`: Contiene información sobre símbolos en ámbitos anteriores (como variables ocultas por una nueva declaración).
- `BufferedWriter out`: Permite escribir en un archivo de salida para guardar el estado de la tabla de símbolos.
- `final String SYMBOLS_TABLE_PATH`: Ruta donde se guarda la tabla de símbolos.
- `String filename`: Nombre del archivo donde se almacena la tabla de símbolos.

Funciones principales

- `initializeFileWriter()`: Abre un archivo para escribir la tabla de símbolos.
- `saveTableInFile(String action)`: Guarda en el archivo la tabla de símbolos junto con una acción realizada (ej. agregar una variable).
- `closeSymbolsTableFiles()` y `closeFiles()`: Cierran los archivos correctamente.

Gestión de ámbitos

- `enterBlock()`: Aumenta el ámbito cuando se entra en un nuevo bloque.
- `leaveBlock()`: Disminuye el ámbito cuando se sale de un bloque, eliminando símbolos fuera de alcance.

Gestión de variables y funciones

- `add(String id, Type type)`: Añade un símbolo a la tabla de descripciones (`descriptionTable`). Si ya existe en el mismo ámbito o es una función/palabra reservada, lanza un error.
- `addParam(String idFun, String idParamBack, String idParam, Type type)`: Añade un parámetro a una función en `expansionTable` y lo asocia en `descriptionTable`.
- `get(String id)`: Devuelve el tipo de un identificador, buscándolo en `descriptionTable` o `expansionTable`.
- `getNumParams(String idFun)`: Devuelve el número de parámetros de una función.
- `getParam(String idFun, int index)`: Devuelve el tipo del parámetro en una función según su posición.

Gestión de errores

Clase CompilerError

La clase CompilerError gestiona los errores en el compilador, dividiéndolos en tres tipos: léxicos, sintácticos y semánticos. Además, los errores se escriben en archivos de texto correspondientes para cada tipo.

Funcionalidades clave:

Tipos de errores (ErrorType):

- LEXICAL: Errores relacionados con la fase de análisis léxico (por ejemplo, tokens no válidos).
- SYNTAX: Errores de sintaxis en el código (por ejemplo, estructuras incorrectas).
- SEMANTIC: Errores semánticos, como el uso incorrecto de variables o tipos.

Constructores:

- El constructor principal recibe un mensaje de error, el tipo de error (ErrorType), y un nombre de archivo. Dependiendo del tipo de error, se utiliza un BufferedWriter correspondiente para escribir el error en el archivo adecuado.
- Otro constructor específico se utiliza para crear un mensaje de error sintáctico, donde se indica el token erróneo y los tokens esperados, utilizando la clase ComplexSymbol.

Escritura en archivos:

- Los errores se escriben en archivos de texto específicos para cada tipo de error:
 - lexicalError.txt
 - syntaxError.txt
 - semanticError.txt
- Si el archivo no existe, se crea al momento de escribir el error.

Clase SymTabError

La clase SymTabError se encarga de gestionar los errores relacionados con la tabla de símbolos en el compilador, escribiendo estos errores en un archivo específico: symbolTableError.txt.

Funcionalidades clave:

1. Escritura en archivo:

- Al ser lanzada la excepción SymTabError, el mensaje de error se escribe en el archivo symbolTableError.txt dentro de la carpeta de salida (src\output\filename\$).
- Si el archivo no existe, se crea automáticamente. Los errores se añaden al final del archivo debido al parámetro true en el FileWriter, que habilita el modo de "append".

2. Constructor:

- El constructor de la clase recibe un mensaje de error que se escribe en el archivo de errores.
- Utiliza un BufferedWriter para escribir el error, y se asegura de que el archivo esté abierto y en modo de escritura correcta. Luego, se flush() el BufferedWriter para asegurarse de que los datos se escriban inmediatamente.

3. Cierre del archivo:

- El método estático closeFile() se encarga de cerrar el archivo de manera segura una vez que ya no se necesite escribir más errores.
- Si el archivo no está abierto, no realiza ninguna acción.

Generación de código intermedio

Código de tres direcciones (C3@)

El lenguaje intermedio C3@ es muy versátil porque combina características de lenguajes de alto nivel, que son fáciles de entender, con elementos de lenguajes de bajo nivel. Gracias a esto, es más sencillo convertirlo a código ensamblador y así lograr que el programa funcione en cualquier tipo de computadora sin depender de su sistema específico.

Una de las características más importantes de C3@ es su estructura simple y clara, que se basa en cuatro partes principales: la operación que se va a realizar, el primer operador, el segundo operador si es necesario, y el lugar donde se guardará el resultado. Esto hace que el lenguaje sea fácil de interpretar y eficiente a la hora de convertirlo en instrucciones que entienda la máquina.

```
public InstructionC3A(Code opCode, String op1, String op2, String dest) {  
    this.opCode = opCode;  
    this.op1 = op1;  
    this.op2 = op2;  
    this.dest = dest;  
}
```

Conjunto de instrucciones

❖ Operaciones aritméticas y lógicas

- Copy: copiar/asignar
- Add: sumar
- Sub: restar
- Prod: multiplicar
- Div: dividir
- Mod: producto o módulo
- Neg: negar
- and: operación AND
- or: operación OR
- not: operación NOT

❖ Valores para operaciones relacionales

- LT (less than): menor estricto

- LE (less equal): menor o igual
- EQ (equal): igual
- NE (not equal): diferente
- GE (greater equal): mayor o igual
- GT (greater than): mayor estricto
- ❖ **Operaciones de salto y etiquetas**
 - skip: ignorar
 - go_0to: salto incondicional
 - jump_cond: salto condicional
- ❖ **Llamadas a procedimientos**
 - pmb: preámbulo
 - call: llamada
 - rtn: retorno
 - param: parámetro
- ❖ **Operaciones de entrada y salida**
 - output: mostrar por pantalla
 - input: introducir por teclado

Con todo definido podemos crear instrucciones con el constructor mostrado a continuación.

```
// Add a new instruction
public void generateC3aInstr(InstructionC3A.Code opCode, String op1, String op2, String dest) {
    InstructionC3A inst = new InstructionC3A(opCode, op1, op2, dest);
    instructions.add(inst);
}
```

Se incorpora a la lista ArrayList para luego guardarlo en un archivo de salida.

Clase BackTables

La clase BackTables se utiliza para gestionar las tablas de procedimientos, variables y etiquetas dentro del proceso de generación de código intermedio. Su función es almacenar y administrar estos elementos, asignando nombres únicos, direcciones y tamaños de memoria, permitiendo su fácil acceso y modificación. Además proporciona métodos para agregar variables, procedimientos y etiquetas. También tiene métodos para generar un fichero de texto.

Para cada una de las tablas hemos creado un objeto para facilitar el acceso a la información de cada una.

Clase Var

Esta clase contiene la información que contendrá cada objeto de la tabla de variables. Esas variables estarán formadas por los siguientes atributos:

- String varName; // Variable name
- int parentId; // Parent ID
- int displacement; // Offset/Displacement
- int allocatedSize; // Space occupation/Allocated size
- TipoSubyacente dataType; // Data type
- boolean isParameter; // Is it a parameter?
- String strValue; // Only used for string variables

En la clase también se definen métodos para acceder a estos atributos y calcular la dirección en ensamblador.

Clase Proc

Esta clase contiene la información que contendrá cada objeto de la tabla de procedimientos.

Esos procedimientos estarán formados por los siguientes atributos:

- String procName; // Procedure name
- TipoSubyacente returnType; // Return type (type)
- int memorySize; // Memory used (size)
- int variableNumber; // Variable number (nv)
- int numParameters; // Number of parameters
- int parametersOffset; // Parameters offset (offset)
- int nextVariableId = 0; // More descriptive name

La clase también permite gestionar la asignación de memoria y los identificadores de variables de cada procedimiento.

Clase Etq

Esta clase contiene la información que contendrá cada objeto de la tabla de etiquetas. Estas etiquetas están formadas por los siguientes atributos:

- String name;

Esta clase es para crear las diferentes etiquetas que van apareciendo al generar el código de tres direcciones.

Generación de código ensamblador

Lenguaje empleado

Como opciones de lenguaje ensamblador destino teníamos dos opciones Easy68k o GAS - GNU Assembler. Nos decantamos por la última debido a dos factores: Aunque todos habíamos tenido que realizar programas en 68k, todos nos encontrábamos un poco oxidados y teniendo en cuenta de que en los apuntes de la asignatura y a lo largo del curso se ha trabajado en GAS, hemos optado por esta última opción.

GAS (GNU Assembler) es el ensamblador oficial del proyecto GNU y el back-end por defecto del GNU Compiler Collection, utilizado para compilar Linux y otros sistemas operativos GNU por lo tanto es un lenguaje altamente adoptado y con muchos recursos online.

Estructura del programa

La generación del programa se encuentra dividido en 3 secciones: Cabecera (*header*), bloque principal y bloque final (*footer*). Este diseño permite una generación modular de código, separando claramente la declaración de datos, la lógica principal del programa, y las rutinas auxiliares necesarias.

Cabecera (*header*)

En esta primera parte se declaran las librerías externas que vamos a emplear. En nuestro caso printf y scanf que son funciones de C que nos permiten realizar la entrada y salida de manera mucho más sencilla. Por otro lado también se declaran las variables globales y los strings que vamos a emplear.

Bloque principal

Después de escribir los datos correspondientes de la cabecera, iteramos sobre el conjunto de instrucciones de 3 direcciones (C3A) generando el código ensamblador equivalente mediante un sistema de mapero. Cada tipo de código de 3 direcciones tiene su *handler* que lo transforma en código ensamblador dónde se realizan los siguientes funciones:

1. Traducción de operaciones:
 - Aritméticas (add, sub, mul, div) usando registros EDI/ESI
 - Comparaciones (CMP_LT, CMP_GT, etc.) con funciones auxiliares
 - Lógicas (and, or, not) con operaciones a nivel de bits
2. Manejo de funciones:
 - Prólogos de funciones (pmb) con ajuste de pila
 - Paso de parámetros (param) mediante push/pop
 - Llamadas (call) con limpieza de registros
3. Control de flujo:
 - Saltos condicionales (jump_cond) usando cmp y je/jg

- Etiquetas para control de bucles y condicionales

Bloque final (footer)

Este bloque final contiene las funciones auxiliares requeridas durante la generación del código ensamblador para, por ejemplo, realizar comparaciones y también funciones de ayuda para el output de variables booleanas.

Optimizaciones código ensamblador

Las optimizaciones que hemos realizado son relativamente sencillas, Por una parte hemos optimizado el acceso a la pila mediante alineación a 16 bytes en prologólogos de funciones.

```
Proc proc = backend.getProc(backFunId);  
int alignedSize = (proc.getMemorySize() + 15) & ~15;  
writeLine("sub $" + alignedSize + ", %rsp");
```

Por otro lado, un sistema inteligente que solo incluye en el footer las funciones comparadoras (CMP_*) y rutinas de impresión necesarias, detectando su uso durante la generación del código ensamblador en el bloque principal.

Conclusión

Para concluir, nos gustaría dar nuestra visión sobre el proyecto.

Por una parte, esta práctica nos ha ayudado a consolidar todos los conceptos sobre la asignatura de Compiladores, y a poner en práctica los conocimientos aportados durante las clases teóricas y prácticas de la asignatura.

También nos gustaría comentar que el trabajo y sacrificio que ha conllevado esta práctica no ha sido leve, todos los compañeros han aportado su granito de arena y el reto ha sido grande.

Consideramos que la práctica también ha sido necesaria para que los ingenieros del grupo consoliden todos los conocimientos tan necesarios sobre computación y la base de generación de código.

No hemos conseguido llegar a tiempo a realizar importantes optimizaciones en código ensamblador, pero entendemos que llegados a este punto, y el esfuerzo que nos ha supuesto, estamos orgullosos de haber creado nuestro primer compilador funcional.

Anexos

Anexo 1: Gramática del lenguaje

Aquí encontramos la gramática del lenguaje sin ver las funciones semánticas que se realizan.

Producción inicial

start with STARTS;

STARTS ::= INIT declare_functions

Inicializar tipos

INIT ::= λ

Inicialización y declaración de funciones

declare_functions ::= define_function declare_functions
| main_s

define_function ::= function_init LBRACE block_in function_instructions function_return block_out
RBRACE

function_init ::= FUNC ID function_params RPAREN TWO_POINTS ID

function_params ::= function_w_params

function_w_params ::= function_w_params COMMA ID TWO_POINTS ID
| LPAREN ID TWO_POINTS ID

function_instructions ::= instructions
| λ

function_return ::= RETURN value SEMICOLON
| EMPTY

Main

main_s ::= FUNC MAIN m_main LPAREN RPAREN LBRACE block_in main_instructions block_out
RBRACE

m_main ::= λ

main_instructions ::= instructions
| EMPTY

Instrucciones

instructions ::= instructions instruction
| instruction

instruction ::= declarations SEMICOLON
| tuple_declaration SEMICOLON

| instruction_if
| instruction_while
| instruction_for
| operator_assign SEMICOLON
| instruction_out SEMICOLON
| call_function SEMICOLON

Marcadores

block_in ::= λ

block_out ::= λ

declarations ::= ID COMMA declarations
| declaration

Tuplas

tuple_declaration ::= STRUCT LBRACE field_list RBRACE ID

field_list ::= field_list SEMICOLON tuple_field
| tuple_field

tuple_field ::= ID TWO_POINTS ID

declaration ::= ID constant TWO_POINTS ID OP_ASSIGN value
| ID constant TWO_POINTS ID OP_ASSIGN tuple_value

Constantes

constant ::= CONSTANT
| EMPTY

Asignación

operator_assign ::= ID OP_ASSIGN value

Operaciones aritméticas

arith_op ::= OP_ARITHMETICAL_B
| OP_ARITHMETICAL_C

arithmetical_operation ::= arithmetical_value arith_op arithmetical_value
| OP_ARITHMETICAL_B LPAREN arithmetical_operation RPAREN
| LPAREN arithmetical_operation RPAREN

number_value ::= NUMBER
| LPAREN number_value RPAREN
| OP_ARITHMETICAL_B number_value

arithmetical_value ::= number_value
| id_value
| arithmetical_operation
| call_function
| OP_ARITHMETICAL_B call_function

Valores

id_value ::= ID

| LPAREN id_value RPAREN
| OP_LOGICAL_NOT id_value
| OP_ARITHMETICAL_B id_value

string_value ::= STRING
| LPAREN string_value RPAREN

value ::= id_value
| string_value
| boolean_value
| number_value
| value_comparison
| OP_LOGICAL_NOT value_comparison
| arithmetical_operation
| boolean_operation
| call_function
| READ LPAREN RPAREN

tuple_value ::= LBRACKET tuple_values RBRACKET

tuple_values ::= tuple_values COMMA value
| value:value

Operaciones booleanas

boolean_operation ::= boolean_op_value OP_LOGICAL boolean_op_value
| LPAREN boolean_operation RPAREN
| OP_LOGICAL_NOT LPAREN boolean_operation RPAREN

boolean_op_value ::= boolean_value
| id_value
| boolean_operation
| value_comparison
| OP_LOGICAL_NOT value_comparison
| call_function
| OP_LOGICAL_NOT call_function

boolean_value ::= BOOL
| OP_LOGICAL_NOT boolean_value
| LPAREN boolean_value RPAREN

value_comparison ::= LPAREN value OP_RELATIONAL value RPAREN
| LPAREN value_comparison RPAREN

Condiciones

cond ::= boolean_op_value

cond_t ::= λ

cond_f ::= λ

Instrucción If

instruction_if ::= IF cond LBRACE block_in cond_t instructions block_out RBRACE instruction_elif
instruction_else

m_end ::= EMPTY

instruction_elif ::= m_end ELIF cond_f cond LBRACE block_in cond_t instructions block_out RBRACE
instruction_elif
| λ

instruction_else ::= m_end ELSE LBRACE block_in cond_f instructions block_out RBRACE
| cond_f

Instrucción While

m_while ::= EMPTY

Instrucción For

instruction_while ::= WHILE m_while cond LBRACE block_in cond_t instructions block_out RBRACE

instruction_for ::= FOR LPAREN for_body RPAREN LBRACE block_in cond_t instructions block_out
RBRACE

label_for ::= EMPTY

for_body ::= declaration SEMICOLON label_for cond SEMICOLON label_post_for operator_assign

label_post_for ::= EMPTY

Llamada a una función

call_function ::= ID LPAREN RPAREN
| call_body RPAREN

call_body ::= call_body COMMA value
| ID LPAREN value

instruction_out ::= PRINT LPAREN value RPAREN

EMPTY ::= λ

Anexo 2: Ejemplos de programas

En el proyecto compartido encontramos 7 archivos de ejemplos de programas. Se encuentran en la carpeta 'src/examples/' y contienen 4 casos con errores (uno para cada tipo de error) y 3 programas exitosos, los dos primeros introductorios, y un último con todas las características implementadas.

Ejemplos exitosos

1. 'example1.txt': En este primer ejemplo encontramos la declaración de los distintos tipos de variables que permite nuestro lenguaje (number, string y boolean). Se asigna un valor de cada tipo a una variable distinta, y se pinta mediante la función 'print'.
2. 'example2.txt': En este segundo caso, encontramos un 'for' a modo de contador y posteriormente una definición de una tupla (persona) que tiene 3 atributos (edad, soltero, altura). Para las tuplas solamente se permite la definición de estas, y la declaración de variables de este tipo. No se ha conseguido generar el acceso a las tuplas.
3. 'example3.txt': Este último ejemplo de programa 100% compilable y ejecutable contiene ejemplos de todas las estructuras que se pueden utilizar en nuestro lenguaje. En primer lugar, encontramos las definiciones de 2 funciones ('fizzbuzz' y 'factorial').

La primera, es la función típica de pruebas de programación, donde en una lista de números pinta fizz si el número es divisible entre 3, buzz si es divisible entre 5, y fizzbuzz si es divisible por ambos números.

La segunda función, factorial, es una función recursiva que permite calcular el factorial de un número (solamente permite calcular el factorial de un valor entre 0 y 15, ya que si es más grande el número, la memoria no gestiona correctamente un valor tan grande).

Tras la definición de las funciones anteriores, pasamos al cuerpo del programa, donde encontramos primero un bucle 'for' para hacer uso de la función 'fizzbuzz', A continuación se solicita un número al usuario para calcular su factorial. Posteriormente, encontramos el contador de una bomba (3, 2, 1, PUM), y finalmente, un módulo que verifica el uso de la estructura condicional 'if-elif-else' que dice si la suma de dos números es un número pequeño, mediano, o grande.

Con todo esto, creemos que es un ejemplo bastante completo de todo lo que puede hacer nuestro lenguaje de programación.

Ejemplos erróneos

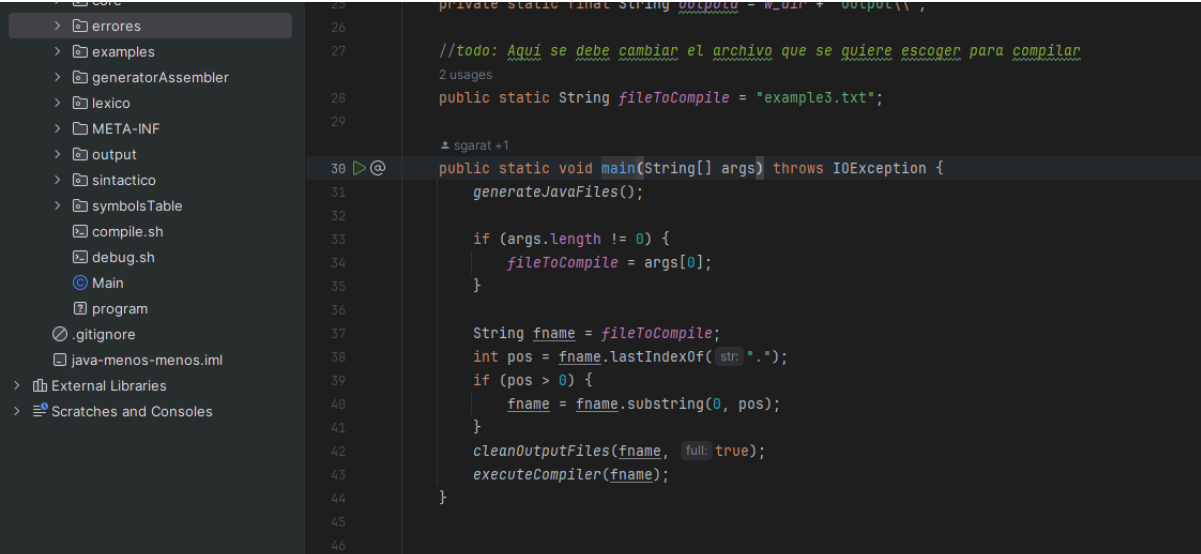
1. 'error1.txt': Este primer ejemplo muestra un error léxico, ya que nuestro lenguaje no permite caracteres especiales en los identificadores.
2. 'error2.txt': El segundo, muestra un error sintactico en la estructura del bucle 'for', ya que nuestro 'for' necesita de una post-instrucción en su esquema.
3. 'error3.txt': El tercero muestra un error semántico, donde los tipos subyacentes no coinciden, y por lo tanto no se permite la asignación (num : number := "HOLA MUNDO!";)
4. 'error4.txt': Este último ejemplo hace saltar un error de la tabla de símbolos, ya que no reconoce la función fizbuz.

Anexo 3: Guía de uso

En este apartado veremos un ejemplo de como correr el proyecto, ejecutar el compilador y el programa resultante.

1.º paso:

Descargamos el proyecto JAVA y en la clase 'Main' encontramos el atributo donde cambiar el fichero que queremos compilar. Si se ejecuta el main por consola, se puede escribir como argumento el fichero a compilar (exampleX.txt o errorX.txt)



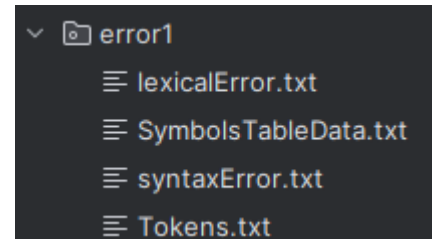
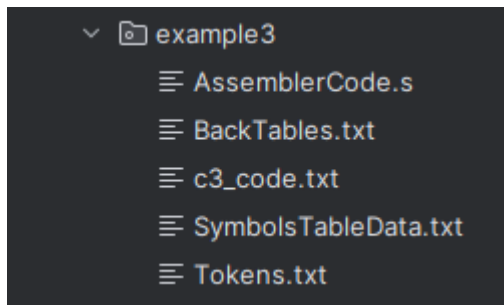
```
private static final String output = "out" + File.separator + "output\\";

//todo: Aqui se debe cambiar el archivo que se quiere escoger para compilar
2 usages
public static String fileToCompile = "example3.txt";

+ sgarat+1
38 > @ public static void main(String[] args) throws IOException {
39     generateJavaFiles();
40
41     if (args.length != 0) {
42         fileToCompile = args[0];
43     }
44
45     String fname = fileToCompile;
46     int pos = fname.lastIndexOf(str: ".");
47     if (pos > 0) {
48         fname = fname.substring(0, pos);
49     }
50     cleanOutputFiles(fname, full: true);
51     executeCompiler(fname);
52 }
```

En este método main primero se hacen las llamadas pertinentes para generar las clases de JFlex y CUP, posteriormente se limpia el directorio de salida y finalmente se ejecuta el compilador, que se encargará de hacer la compilación y la generación de los archivos

resultantes; en un caso exitoso generará los Tokens, el código de 3 direcciones, el código ensamblador, y los ficheros de las tablas del Backend (tabla de variables, etiquetas y procedimientos) y la tabla de símbolos. Por otro lado, en un caso erróneo, se generarán los archivos de salida que se hayan podido generar (como los Tokens identificados hasta el error) y los ficheros de error correspondientes. Estos se encontrarán en la carpeta “src/output/archivoCompilado/”



2.º paso:

Una vez compilado el programa introducido por el usuario, y el archivo del código ensamblador generado, el usuario deberá ejecutar por consola el comando `./compile.sh`, el cual dará el resultado de la ejecución del programa. Este archivo se encuentra en el directorio “src/”. Cabe destacar que para poder hacer uso de este script de comandos, se debe hacer sobre una consola de linux y tener instalado el compilador ‘gcc’

(Revisar `gcc -version` o sino instalar con ‘`sudo apt update && sudo apt install gcc -y`’)

```
#!/bin/sh
rm -f ./program
gcc -no-pie ./output/example3/AssemblerCode.s -o program
FILE=./program
if [ -f "$FILE" ]; then
    $FILE
fi
```

Tras esto, se habrá generado el archivo ‘program.s’ y se ejecutará para mostrarnos el resultado de la ejecución.

```
sergio@564SERGIO:/mnt/c/Users/sergi/Desktop/Compiladores/lastCompiler/java-menos-menos/src$ ./compile.sh
/usr/bin/ld: warning: /tmp/ccZtSolv.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
1:
2:
3: fizz
4:
5: buzz
6: fizz
7:
8:
9: fizz
10: buzz
11:
12: fizz
13:
14:
15: fizzbuzz
16:
17:
18: fizz
19:
20: buzz
Inserta un número (entre 0 y 15) para calcular el factorial: 5
Número insertado: 5
5! = 120
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, PUM
Inserta un número a: 3455
Número insertado: 3455
Inserta un número b: 234
Número insertado: 234
3689: Es un número grande
HE ACABADO
sergio@564SERGIO:/mnt/c/Users/sergi/Desktop/Compiladores/lastCompiler/java-menos-menos/src$
```

Anexo 4: Vídeo explicativo

Se encuentra subido a YouTube el video explicativo realizado por todos los miembros explicando la práctica y con un ejemplo de ejecución

<https://www.youtube.com/watch?v=k1X0oU7Xts0>